ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΚΡΥΠΤΟΓΡΑΦΙΑ

ΣΥΝΟΛΙΚΟΙ ΠΟΝΤΟΙ: 720

Username for CryptoHack: kolovos1477

ΚΟΛΟΒΟΣ ΚΥΡΙΑΚΟΣ, Π19196
kyriakos77kolovos@gmail.com

Challenges

▪ INTRODUCTION

A. Introduction

1. Finding flags (2 points)

Each challenge is designed to help introduce you to a new piece of cryptography. Solving a challenge will require you to find a "flag".

These flags will usually be in the format **crypto{y0ur_f1rst_fl4g}**. The flag format helps you verify that you found the correct solution.

Try submitting this into the form below to solve your first challenge.

FLAG: crypto{y0ur_f1rst_fl4g}

2. Great Snakes (3 points)

Modern cryptography involves code, and code involves coding. CryptoHack provides a good opportunity to sharpen your skills.

Of all modern programming languages, Python 3 stands out as ideal for quickly writing cryptographic scripts and attacks. For more information about why we think Python is so great for this, please see the FAQ.

Run the attached Python script and it will output your flag.

FLAG: crypto{z3n_0f_pyth0n}

3. Networks Attacks (5 points)

Several of the challenges are dynamic and require you to talk to our challenge servers over the network. This allows you to perform man-in-the-middle attacks on people trying to communicate, or directly attack a vulnerable service. To keep things consistent, our interactive servers always send and receive JSON objects.

Python makes such network communication easy with the telnetlib module. Conveniently, it's part of Python's standard library, so let's use it for now.

For this challenge, connect to socket.cryptohack.org on port 11112. Send a JSON object with the key buy and value flag.

The example script below contains the beginnings of a solution for you to modify, and you can reuse it for later challenges.

Connect at nc socket.cryptohack.org 11112

Solving the Challenge:

```
b"Welcome to netcat's flag shop!\n"
b'What would you like to buy?\n'
b"I only speak JSON, I hope that's ok.\n"
b'\n'
b'{"buy": "clothes"}'
{'error': 'Sorry! All we have to sell are flags.'}
```

Running the given python file that CryptoHack provided us we are getting an "error" and not the results that we want to get.

In order for us to get the flag we must change the value of the JSON object from "clothes" to "flag". Running the code, the server returns the flag.

```
request = {
    "buy": "flag"
}
```

```
request = {
    "buy": "clothes"
}
```

```
b"Welcome to netcat's flag shop!\n"
b'What would you like to buy?\n'
b"I only speak JSON, I hope that's ok.\n"
b'\n'
b'{"buy": "flag"}'
{'flag': 'crypto{sh0pp1ng_f0r_fl4g5}'}
```

FLAG: crypto{sh0pp1ng_f0r_fl4g5}

- GENERAL

B. ENCODING

1. ASCII (5 points)

ASCII is a 7-bit encoding standard which allows the representation of text using the integers 0-127.

Using the below integer array, convert the numbers to their corresponding ASCII characters to obtain a flag.

[99, 114, 121, 112, 116, 111, 123, 65, 83, 67, 73, 73, 95, 112, 114, 49, 110, 116, 52, 98, 108, 51, 125]

Solving the challenge:

From what we can see from the instruction of the challenge, we are given an array of numbers and we have to convert them to ASCII characters. We are using the join method to create a string from this array.

```
array = [99, 114, 121, 112, 116, 111, 123, 65, 83, 67, 73, 73, 95, 112, 114, 49, 110, 116, 52, 98, 108,
  51, 125]

FLAG = "".join(chr(num) for num in array)
print(FLAG)
```

FLAG: crypto {ASCII_pr1nt4bl3}

2. Hex (5 points)

When we encrypt something the resulting ciphertext commonly has bytes which are not printable ASCII characters. If we want to share our encrypted data, it's common to encode it into something more user-friendly and portable across different systems.

Included below is a flag encoded as a hex string. Decode this back into bytes to get the flag.

63727970746f7b596f755f77696c6c5f62655f776f726b696e675f776974685f6865785f737472696e67735f615f6c6f747d

Solving the challenge:

We have to unhexify the given hex string that we were given. To do so we can import a builtin method from binascii library called unhexify. After we unhexify the hex string we have to decode using the decode method and print the flag.

```python
from binascii import unhexlify

hex_string =
  "63727970746f7b596f755f77696c6c5f62655f776f726b696e675f776974685f68
  65785f737472696e67735f615f6c6f747d"

hex_string = unhexlify(hex_string)
print(hex_string.decode())
```

FLAG: crypto {You_will_be_working_with_hex_strings_a_lot}

3. **Base64 (10 points)**

Another common encoding scheme is Base64, which allows us to represent binary data as an ASCII string using 64 characters. One character of a Base64 string encodes 6 bits, and so 4 characters of Base64 encodes three 8-bit bytes.

Base64 is most commonly used online, where binary data such as images can be easily included into html or CSS files.

Take the below hex string, decode it into bytes and then encode it into Base64.

72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf

Solving the challenge:

From the challenge instructions we have asked to take the hex string, decode it into bytes and then encode it into Base64.
We have used 2 built-in methods, unhexify and b64encode from binascii and base64 libraries. Using the unhexify method we decode the hex string to bytes and then using the b64encode method we encode the bytes to the final flag.

```python
from base64 import b64encode
from binascii import unhexlify

hex_encoded_string = unhexlify("72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf")
encoded_b64 = b64encode(hex_encoded_string)
print(encoded_b64.decode())
```

FLAG: crypto/Base+64+Encoding+is+Web+Safe/

4. **Bytes and Big Integers (10 points)**

Cryptosystems like RSA works on numbers, but messages are made up of characters. How should we convert our messages into numbers so that mathematical operations can be applied?

The most common way is to take the ordinal bytes of the message, convert them into hexadecimal, and concatenate. This can be interpreted as a base-16 number, and also represented in base-10.

To illustrate:

message: HELLO
ascii bytes: [72, 69, 76, 76, 79] hex
bytes: [0x48, 0x45, 0x4c, 0x4c, 0x4f]
base-16: 0x48454c4c4f
base-10: 310400273487

Python's PyCryptoDome library implements this with the methods Crypto.Util.number.bytes_to_long and Crypto.Util.number.long_to_bytes.

Convert the following integer back into a message:

11515195063862318899931685488813747395775516287289682636499965282714637259206269

Solving the challenge:

We have been given a long integer (Big Integer). We have been asked to decode it using a specific method from a library called PyCryptoDome. Using the method long_to_bytes, we pass as a parameter the long integer we have been given and it returns a byte string, which in our case is the FLAG.

```
from Crypto.Util.number import long_to_bytes

print(long_to_bytes(11515195063862318899931685488813747395775516287289682636499965282714637259206269
  ).decode())
```

FLAG: crypto{3nc0d1n6_4ll_7h3_w4y_d0wn}

5. **Encoding Challenge (40 points)**

Now you've got the hang of the various encodings you'll be encountering, let's have a look at automating it.

Can you pass all 100 levels to get the flag?

The 13377.py file attached below is the source code for what's running on the server. The pwntools_example.py file provides the start of a solution using the incredibly convenient pwntools library. which you can use if you like. pwntools however is incompatible with Windows, so telnetlib_example.py is also provided.

For more information about connecting to interactive challenges, see the FAQ. Feel free to skip ahead to the cryptography if you aren't in the mood for a coding challenge!

Connect at nc socket.cryptohack.org 13377

Solving the challenge:

The challenge gives us 2 python files, 13377 which contains the source code that is running on the server and pwntools_example which contains few examples of the library.
We have to create a program that connects with the server and decode the returned value from the server, which is a **JSON object {type: value}**. The types that the server returns are encodings of base64, hex, rot-13, bigint and utf-8. From the challenge instructions we have been told that in order to get the flag we have to pass all the 100 levels.
In the beginning we started requesting one JSON object from the server in order for us to see what the server returns and checking all the possibilities. Even though we have the server source code we have to make sure what we were getting us a result. After that we have created decoders for all the encoding types that the server returned to us and sent back the decoded JSON object. We loop the receive from server and sent back situation 100 times. If the decode was correct the server returned back to us a new JSON object with new type and value. At the end after we passed all 100 levels successfully the server returned a new JSON object {flag: value}.

```python
import codecs
import json
import pwn
from base64 import b64decode

r = pwn.remote('socket.cryptohack.org', 13377)

received_decoded = {
    "decoded": "changeme"
}


def json_recv():
    tmp = r.recvline()
    return json.loads(tmp.decode())


def json_send(json_obj):
    req = json.dumps(json_obj).encode()
    r.sendline(req)
```

First, we import the libraries we want to use: 1. Codecs: is used from decoding rot-13.
2. Json: to load the json object the server returns.
3. Pwn: create connection with the server, retrieve the json object and send back the decoded json object. 4. Base64: to decode base64 strings.

```python
def base64Decoder(encoded_txt):
    decoded_txt = b64decode(encoded_txt).decode()
    return decoded_txt


def hexDecoder(encoded_txt):
    decoded_txt = bytes.fromhex(encoded_txt).decode("utf-8")
    return decoded_txt


def rot13Decoder(encoded_txt):
    decoded_txt = codecs.decode(encoded_txt, 'rot13')
    return decoded_txt


def bigIntDecoder(encoded_txt):
    decoded_txt = hexDecoder(encoded_txt[2:])
    return decoded_txt


def utf8Decoder(encoded_txt):
    decoded_txt = "".join([chr(num) for num in encoded_txt])
    return decoded_txt
```

Methods that are used to decode the encoded objects.

```python
def start():
    for i in range(101):
        received = json_recv()
        print(received)

        if list(received)[0] == "type":
            if received["type"] == "base64":
                received_decoded["decoded"] = base64Decoder(received["encoded"])
            elif received["type"] == "hex":
                received_decoded["decoded"] = hexDecoder(received["encoded"])
            elif received["type"] == "rot13":
                received_decoded["decoded"] = rot13Decoder(received["encoded"])
            elif received["type"] == "bigint":
                received_decoded["decoded"] = bigIntDecoder(received["encoded"])
            elif received["type"] == "utf-8":
                received_decoded["decoded"] = utf8Decoder(received["encoded"])
            print(received_decoded)
            json_send(received_decoded)
        elif list(received)[0] == "flag":
            print("Here's the flag: {}".format(received["flag"]))
        else:
            print("There was an error")


start()
```

Start method is the main method of the program. We loop 100 times, each time retrieving a new json object, decode it based on its type and send back to the server to receive a new one. At the end server sends a new json object with attribute flag and that's where the program stops, printing the flag in the terminal.

FLAG: crypto{3nc0d3_d3c0d3_3nc0d3}

C. XOR

1. XOR Starter (10 point)

XOR is a bitwise operator which returns 0 if the bits are the same, and 1 otherwise. In textbooks the XOR operator is denoted by $\oplus$, but in most challenges and programming languages you will see the caret ^ used instead.

A	B	Output
0	0	0

0	1	1
1	0	1
1	1	0

For longer binary numbers we XOR bit by bit: 0110 ^ 1010 = 1100. We can XOR integers by first converting the integer from decimal to binary. We can XOR strings by first converting each character to the integer representing the Unicode character.

Given the string "label", XOR each character with the integer 13. Convert these integers back to a string and submit the flag as **crypto{new_string}**.

Solving the challenge:

In order for us to find the flag we have to convert each character to its representing integer Unicode character and XOR it with the number 13. The new number that was create by XORing the two numbers is a new character, so now we can use the char method to convert that number to character. After this we create a new string from these new characters. At the end of the conversion, we can print the flag.

```
string = 'label'
flag = ''

# create flag string xoring each char with the number 13
for c in string:
    flag += chr(ord(c) ^ 13)

# print crypto{flag}
print('crypto{{{}}}'.format(flag))
```

FLAG: crypto{aloha}

2. XOR Properties (15 points)

In the last challenge, you saw how XOR worked at the level of bits. In this one, we're going to cover the properties of the XOR operation and then use them to undo a chain of operations that have encrypted a flag. Gaining an intuition for how this works will help greatly when you come to attacking real cryptosystems later, especially in the block ciphers category.

There are four main properties we should consider when we solve challenges using the XOR operator

Commutative: A ⊕ B = B ⊕ A

Associative: A ⊕ (B ⊕ C) = (A ⊕ B) ⊕ C
Identity: A ⊕ 0 = A
Self-Inverse: A ⊕ A = 0

Let's break this down. Commutative means that the order of the XOR operations is not important. Associative means that a chain of operations can be carried out without order (we do not need to worry about brackets). The identity is 0, so XOR with 0 "does nothing", and lastly something XOR'd with itself returns zero.

Let's try this out in action! Below is a series of outputs where three random keys have been XOR'd together and with the flag. Use the above properties to undo the encryption in the final line to obtain the flag.

KEY1 = a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313
KEY2 ^ KEY1 = 37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e
KEY2 ^ KEY3 = c1545756687e7573db23aa1c3452a098b71a7fbf0fdddde5fc1
FLAG ^ KEY1 ^ KEY3 ^ KEY2 = 04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf

SOS: Before you XOR these objects, be sure to decode from hex to bytes. If you have pwntools installed, you have a xor function for byte strings: from pwn import xor

Solving the challenge:
Based on the properties of XOR, we can find very easy the missing keys, key 2 and key 3, and later the flag.
We have been given the key 1. We can use the key 1 and (key2 ^ key1) to find key2 and later the other two keys, key 3 and flag.

```python
from pwn import xor
from binascii import unhexlify

KEY1 = 'a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313'
KEY2 = xor(unhexlify(KEY1), unhexlify('37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e'))
KEY3 = xor(KEY2, unhexlify('c1545756687e7573db23aa1c3452a098b71a7fbf0fdddde5fc1'))

KEY1_KEY2 = xor(unhexlify(KEY1), KEY2)
KEY1_KEY2_KEY3 = xor(KEY1_KEY2, KEY3)

FLAG = xor(KEY1_KEY2_KEY3, unhexlify('04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf'))

print(FLAG.decode('utf-8'))
```

FLAG: crypto{x0r_i5_ass0c1at1v3}

3. **Favorite Byte (20 points)**

I've hidden my data using XOR with a single byte. Don't forget to decode from hex first.

73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d

Solving the challenge:

First of all, we have to understand what a single byte means. A single byte character is a mapping of 256 individual characters to their identifying code values. This means a single byte can be 256 different bytes, so we have to XOR each unhex byte of the given string with that single byte.

```
from binascii import unhexlify

string = unhexlify("73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d")
l = [c for c in string]
for i in range(256):
    a = "".join([chr(n ^ i) for n in l])
    if a.startswith("crypto"):
        print(a)
```

FLAG: crypto{0x10_15_my_f4v0ur173_by7e}

4. **You either know, XOR you don't (30 points)**

I've encrypted the flag with my secret key, you'll never be able to guess it.

SOS: Remember the flag format and how it might help you in this challenge!

0e0b213f26041e480b26217f27342e175d0e070a3c5b103e2526217f27342e175d0e077e263451150104

Solving the challenge:

From the challenge we can see that the message is encrypted by a secret key. We have a hint though, that help us maybe guess it. Without the hint we would have to brute force our way to find that secret key. We know the general format of the flag, crypto{value}. With this we know that the first 7 element of the string are equal with the elements "crypto{". Now we know that these 7 elements are equal with the specific word we have to prove them. Based on the previous challenge we can XOR the first 7 elements of the given hex string with the part of the flag, "crypto{", with the method of brute force. The returned value of this XOR is a part of the secret key, "myXORke". Though we are missing one character, "y", to correct it. Now we know that the secret key is "myXORkey". Now we have to make the length of the secret to be the same with the hex string. At the end we can XOR the hex string with the encoded secret key. The result is the flag.

```python
from binascii import unhexlify
from pwn import xor


def bruteforce(encoded, key):
    tmp = b''

    for i in range(len(encoded)):
        tmp += xor(encoded[i], key[i])

    return tmp.decode()
```

Brute force method that XORs each character of the encoded text and secret key.

```python
# FLAG format = crypto={}
cipher_txt = unhexlify('0e0b213f26041e480b26217f27342e175d0e070a3c5b103e2526217f27342e175d0e077e263451150104')

# maybe secret_key = crypto{
part_of_FLAG = 'crypto{'

print(bruteforce(cipher_txt[:7], part_of_FLAG))
# chiper_txt[:7] because crypto={ are the first 7 elements of
secret_key = (bruteforce(cipher_txt[:7], part_of_FLAG) + 'y')

# secret_key length must be equal to the cipher text
secret_key += secret_key * int((len(cipher_txt) - len(secret_key)) / len(secret_key))  # 40 lenght instead of 42
# so we adding 2 more to the secret_key
secret_key += secret_key[:2]

FLAG = bruteforce(cipher_txt, secret_key.encode())
print(FLAG)
```

FLAG: crypto{1f_y0u_Kn0w_En0uGH_y0u_Kn0w_1t_4ll}

5. Lemur XOR (40 points)

I've hidden two cool images by XOR with the same secret key so you can't see them!

Solving the challenge:

We have been given 2 PNG files. We have to XOR them in order for us to get a new PNG file with the flag. A color is a combination of 3 basic colors RGB, red, green and blue. In order for us to create the new PNG file we have to XOR each pixel of the one PNG file with the other. We save the results into an array. The new array is used to create the new PNG file.

```python
from PIL import Image

lemur = Image.open("/Users/bakeries/Desktop/Crypto_Exam/From_Me/General/XOR/IMAGES/lemur.png")
flag = Image.open("/Users/bakeries/Desktop/Crypto_Exam/From_Me/General/XOR/IMAGES/flag.png")

pixels_lemur = lemur.load()  # create the pixel map
pixels_flag = flag.load()

for i in range(lemur.size[0]):  # for every pixel:
    for j in range(lemur.size[1]):
        # Gather each pixel
        l = pixels_lemur[i, j]
        f = pixels_flag[i, j]

        # XOR each part of the pixel tuple
        r = l[0] ^ f[0]
        g = l[1] ^ f[1]
        b = l[2] ^ f[2]

        # Store the resulatant tuple back into an image
        pixels_flag[i, j] = (r, g, b)

flag.save("lemur_xor_flag.png")
```

FLAG: crypto{X0Rly_not!}

- # MATHEMATICS

BRAINTEASERS PART 2:

1. **Cofactor Cofantasy (150pts)**

Do you have the power to break out of my cofactor cofantasy?
Calculating the solution should take less than 15 minutes.
Challenge contributed by Robin_Jadoul and Thunderlord
Connect at nc socket.cryptohack.org 13398
13398.py

Solving the challenge:

File Edit Search View Document Help

```python
import math
from Crypto.Util.number import *
import random
from sympy.ntheory.residue_ntheory import *
from pwn import *
import json
#from decimal import *
#getcontext().prec = 100000
# flag = 'crypto{????????????????????????????????????}'
# print(len(flag))
def legendre(a, p):
    # if a == 2:
        # return (-1)**((p**2-1)//8)
    return pow(a, (p-1)//2, p)


N = 56135841374488684373258694423292882709478511628224823806418810596720294684253418942704418179091997825551647866062286502441190115027708222460662070779175994701788428003909010382045613207284532791741836737030-6663311944661040006934585291004296083372192319606579530917382712591915541731339664276321086006063914107384657485406363955651471413285843546871251531407507293917519967989839818282599493632048361019836647267760-1279175661901110892214276223913861744908916933728985019521611326456685526775192453272881595522432288387752704270544165270943070029947281870578422937019846821583702091492817838824887802189076832440189737006245-8534988419833355585910991945068678054200449928276022337884681087044963339861666995150595584452910991635838842242860413523653147421389150679346662540294124801583459015410394782277120793962245915638608030563460-7708050635024963263051486393844588880622395112435509446868253981530945815153111763792782062904260540218875114491227464449869589727 7
phi = 56135841374488684373258694423292882709478511628224823806413974550086974518248002462797814062141189227167574137989180030483816863197632033192968896065500768938801786598807509315219962138010136188406833851300860971268861927441791178122071599752664078796430411769850033154303492519678490546174370674967628006608883921446643391928676612309188944630598436046965165653521059849130029755392547765534845440469855594908670531-4770208158988191269196601566112047847765854691297222775959632881312422902373604131294051453006051581845240562769630249702344302553885828366721479625676429194602087233559163742525061716900585435677320031980602-5383600867249245507854444944247271236512762862928377312636509414635015681059408293599620885666962033325144399907575703493861474848207357564786217896416914273971930250293888191200848596850672050597558452737185-8919538816922894791118416628613269953271567353945147100596946557062443165864432236665368651790800032723897494367584853197467438284 8
phi = 56135841374488684373258694423292882709478511628224823806413974550086974518248002462797814062141189227167574137989180030483816863197632033192968896065500768938801786598807509315219962138010136188406833851300860971268861927441791178122071599752664078796430411769850033154303492519678490546174370674967628006608883921446643391928676612309188944630598436046965165653521059849130029755392547765534845440469855594908670531-4770208158988191269196601566112047847765854691297222775959632881312422902373604131294051453006051581845240562769630249702344302553885828366721479625676429194602087233559163742525061716900585435677320031980602-5383600867249245507854444944247271236512762862928377312636509414635015681059408293599620885666962033325144399907575703493861474848207357564786217896416914273971930250293888191200848596850672050597558452737185-8919538816922894791118416628613269953271567353945147100596946557062443165864432236665368651790800032723897494367584853197467438284 8
#carm = 17131299247585658072893888678983423678429721566230720155765983444240497040551765328362469672061734702049481622281433198497976267082617288796902148637392483333745686641868619736389044610626636317159527134096-8657570460874250542943668276110832295656701551221062312317906320297012929641435124950276946033569537623267043043027321983829440060102731505870375660756119848956252786263963437667164775038281393388548250937052847353564144345480101501942783217726979325772490607536608774683017707184562257120145889725665298946549196053374115752265455024883684778951176390459217616839178733011950428388221929067689774889151977636659555-3644897725600966033253324714577369278343593229015420917862225788422109407330575366199367848723206583408947751085684310725753016607890550683173153577176338927366035470394537084986834620687013262864943015661-2829449101844145138891118355900045762106801560652205536070845506916028798176401439412038773129510611549040983388545182250203686 6
g = 986762276114520220801525811758560961667498483061127810099097 7

t = phi
while t % 2 == 0:
    t //= 2
factors = set()
for b in range(2, 200):
    #print(b)
    common = GCD(pow(b, t, N) - 1, N)
```

```
File  Edit  Search  View  Document  Help
for b in range(2, 200):
    #print(b)
    common = GCD(pow(b, t, N) - 1, N)
    if not common == 1:
        factors.add(common)


print(len(factors))
ffacs = set()
m = 1
for f in factors:
    for ff in factors:
        if not GCD(f, ff) == 1:
            ffacs.add(GCD(f, ff))

#print(ffacs)
facs = set()
for f in facs:
    if isPrime(f):
        facs.add(f)
    #m *= f//GCD(m, f)

print(facs)
facs = [1813228053839407030072659066430448431904897887327851823214 87, 125715921277942030621916921794679721978341572836598850786902 7, 962196273251325220586440176597115791597983910091568285193479,
  180175212133024978935595973878123066209466737635962575170154 3, 128891176934518228097816204797338172238082363567156659351980 7, 167697651065176806741235096468691059341794737918490447823407,
  212765686518092895538628100582302362078634915228047331602231 9, 919868868284782134728028149935550541290069139895377530961 3, 804417174623672415450557634612568265192863476713582219744267,
  295768852056283474252881603240585602326913528207486218745641 9, 760363004025578077604626764282706830072847466932877419051319, 165673941415563241074849654376983571153997421025752932907 47,
  177934021203989377339172600136870977686030628874155908953275 9, 203596074619604799045796945257280586903861604067177056820 5203, 258004840380588586952059465495491820150647619936419874464 6143,
  119865977856684287465687679965068263670232959251561287389806 7]
facs = list(facs)
val = facs[1] #factor selected such that legendre (generator, factor) == 1
flag = ''
for i in range(43):
    bits = ''
    for ii in range(8):
        allones = True
        while True:
            try:
                r = remote('socket.cryptohack.org', 13398)
                r.recvline()
                for j in range(10):
                    req = {"option":"get_bit", "i":i*8 + ii}
                    r.sendline(json.dumps(req))
                r.recvline()
                for j in range(10):
                    req = {"option":"get_bit", "i":i*8 + ii}
                    r.sendline(json.dumps(req))
                    resp = json.loads(r.recvline())
                    resp = int(resp["bit"], 16)
                    if not legendre_symbol(resp, val) == 1:
                        allones = False
                        break
                break
            except KeyboardInterrupt:
                break
            except:
                continue
        bits += str(int(allones))
    #print(flag)
    flag += chr(int(bits[::-1], 2))
    print(flag)
```

FLAG: crypto{0ver3ng1neering_ch4lleng3_s0lution$}

- RSA

SIGNATURES PART 2:

1. **Vote for Pedro (150pts)**

   If you want my flag, you better vote for Pedro! Can you sign your vote to the server as Alice?

   Connect at nc socket.cryptohack.org 13375

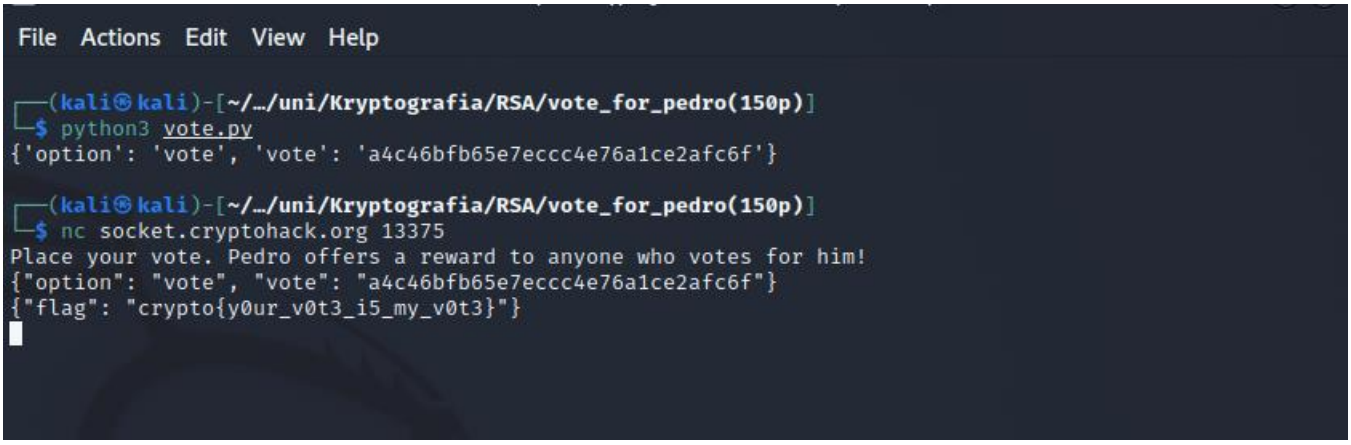
   13375.py

   alice.key

Solving the challenge:

I used this program to generate the JSON so we can vote for Pedro.

```
from Crypto.Util.number import bytes_to_long, long_to_bytes, inverse
from random import randint

N = 22266616657574989868109324252160663470925207690694094953312891282341426880506924648525181014287214350136557941201445475540830225059514652125310445352175047408966028497316806142156338927162621004774769949534239479839334209147097793526897976241752644573955277203987656815646922449168203031499488024798333296412175930765827008394700546657807715318520619975956990281083211405881847851847071572606496061748291017203574300353812240440142861494899725720505181663738931151677884218457824676140190841393217857683627886497104915390385283364971133316672332846071665082777884028170668140862010444247560019193505999704028222347577
s = bytes_to_long(b'VOTE FOR PEDRO')

tot = 2**(8*15-1)

d = inverse(3, tot)

val = pow(s, d, 2**(8*15))

vote = {"option":"vote", "vote":hex(val)[2:]}

print(vote)
```



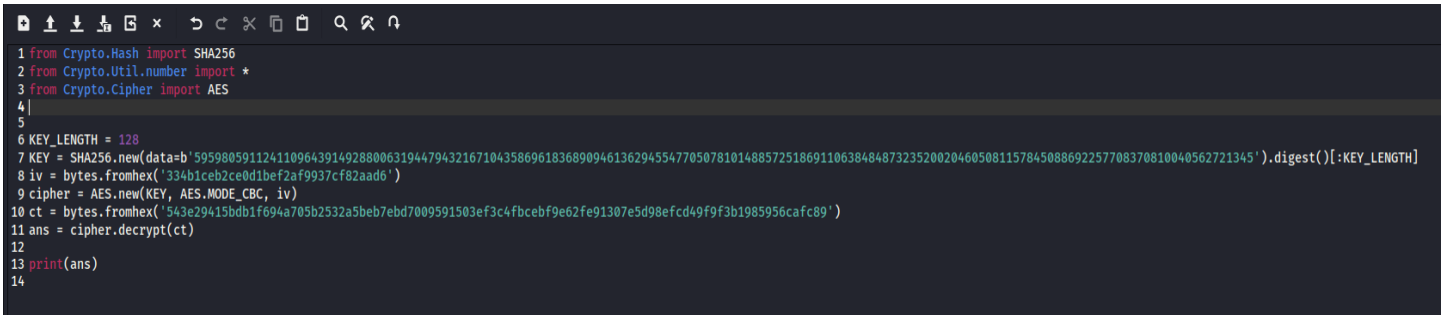
FLAG: crypto{y0ur_v0t3_i5_my_v0t3}

- DIFFIE-HELLMAN

MISC:

1. **The Matrix Reloaded (100pts):**
   It's happening exactly as before... Well, not exactly.
   matrix_reloaded.zip
   Challenge contributed by Jschnei

Solving the challenge:



```
from Crypto.Hash import SHA256
from Crypto.Util.number import *
from Crypto.Cipher import AES

KEY_LENGTH = 128
KEY = SHA256.new(data=b'595980591124110964391492880063194479432167104358696183689094613629455477050781014885725186911063848487323520020460508115784508869225770837081004056272134 5').digest()[:KEY_LENGTH]
iv = bytes.fromhex('334b1ceb2ce0d1bef2af9937cf82aad6')
cipher = AES.new(KEY, AES.MODE_CBC, iv)
ct = bytes.fromhex('543e29415bdb1f694a705b2532a5beb7ebd7009591503ef3c4fbcebf9e62fe91307e5d98efcd49f9f3b1985956cafc89')
ans = cipher.decrypt(ct)

print(ans)
```

Basic explanation on how to get number
Get Jordan normal form of generator (Matrices J and P such that G = P*J*P^-1)
Realize that J is **almost** diagonal except for last 2 which form a 2x2 Jordan Block
Let this Jordan block be K = [[E, 1], [0, E]]
K^n = [[E^n, n*E^(n-1)], [0, E^n]]
Let A = P^-1*w
Let B = P^-1 * v
K^n * [B[-2], B[-1]] = [A[-2], A[-1]]
A[-1] = B[-1]*E^n
A[-2] = B[-2]*E^n + B[-1]*n*E^(n-1)
A[-2] = A[-1]/B[-1]*B[-2] + A[-1]*n/E
n = (A[-2] - A[-1]/B[-1]*B[-2])E/A[-1]

```
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/…/uni/Kryptografia/Diffie-Hellman/matrix_reloaded(100p)]
└─$ python3 matrix.py
b'crypto{the_oracle_told_me_about_you_91e019ff}\x03\x03\x03'
```

FLAG: crypto{the_oracle_told_me_about_you_91e019ff}

2. **The Matrix Revolutions (125pts):**
   Everything that has a beginning has an end, Neo.
   matrix_revolutions.zip
   Challenge contributed by Jschnei

Solving the challenge:

I made this program to generate the MAT_STR.txt binary that I used.

```sage
def load_matrix(fname, base):
    data = open(fname, 'r').read().strip()
    rows = [list(map(int, row)) for row in data.splitlines()]
    #rows = [list(map(F, row)) for row in rows]
    #print(rows)
    MSpace = MatrixSpace(base, 150, 150)
    return sage.matrix.matrix_generic_dense.Matrix_generic_dense(MSpace, rows, True, True)

def get_identity(v, base):
    MSpace = MatrixSpace(base, 150, 150)
    rows = []
    for i in range(150):
        zeros = [0 for _ in range(150)]
        zeros[i] = v
        rows.append(zeros)
    return sage.matrix.matrix_generic_dense.Matrix_generic_dense(MSpace, rows, True, True)


def get_inverse(M, base):
    MI = M.augment(get_identity(1, base)).rref()
    MI = MI.submatrix(0, 150, 150, 150)
    return MI

def ktheigen(M, k):
    Mroots = M.charpoly().roots()
    count = 0
    last = None
    for v, n in Mroots:
        if count > k:
            break
        last = v
        count += n
    return last

def eigenvectors(M, ev, base):
    Meig = M - get_identity(ev, base)
    v = []
    # Etest = Mext - get_identity(ktheigen(Mext, z))
    Etestec = copy(Meig.echelon_form())
    for i in range(149):
        rv = Etestec.column(150-1)[-i-2]
        rs = Etestec.column(150-i-2)[-i-2]
        Etestec.add_multiple_of_column(150-1, 150-i-2, rv/rs)
        v.insert(0, rv/rs)

```

```
    return vector(v + [1])

def random_vector(F):
    v = []
    for i in range(150):
        v.append(F.random_element())
    return v

def get_matrix(rows, base, size=150):
    MSpace = MatrixSpace(base, size, size)
    return sage.matrix.matrix_generic_dense.Matrix_generic_dense(MSpace, rows, True, True)


def basis(w, base):
    vspace = [w] + [[0 for i in range(150)] for _ in range(149)]
    # MSpace = MatrixSpace(GF(2^base), 150, 150)
    M = get_matrix(vspace, base)
    for i in range(149):
        while (True):
            r_v = random_vector(base)
            vspace[i+1] = r_v
            temp = get_matrix(vspace, base)
            if (temp.rank() == i+2):
                # M = M.insert_row(0, r_v)
                break
            print(i)
        print(i)
    return vspace


def transpose(rows, base):
    new_rows = []
    for i in range(150):
        row = []
        for j in range(150):
            row.append(rows[j][i])
        new_rows.append(row)
    return new_rows

```

```
def mat_mul(M1, M2, base):
    cols = []
    for i in range(150):
        cols.append(M1*M2.column(i))
    return get_matrix(transpose(cols, base), base)


def to_pari_poly(poly, v):
    pari_poly = ''
    for k in poly.dict().keys():
        pari_poly += f'Mod(1, 2)*{v}^{k}+'

    return pari_poly[:-1]


anss = []
moduli = []
for ii, vv in enumerate([61, 89]):

    M2 = load_matrix('generator.txt', GF(2))

    modulus = M2.charpoly().factor()[ii][0]
    BASIS = GF(2^vv, 'e', modulus=modulus)
    pari_poly = to_pari_poly(modulus, 'x')
    # for k in modulus.dict().keys():
    #     pari_poly += f'Mod(1, 2)*x^{k}+'

    # pari_poly = pari_poly[:-1]

    print(pari_poly)
    print(BASIS)

    M = load_matrix('generator.txt', BASIS)
    A_pub = load_matrix('alice.pub', BASIS)

    print('loaded')
    eigen_vecs = eigenvectors(M, ktheigen(M, 0), BASIS)
    # print(eigen_vecs)
    P = get_matrix(transpose(basis(eigenvectors(M, ktheigen(M, 0), BASIS), BASIS), BASIS), BASIS)
    K = mat_mul(get_inverse(P, BASIS), M, BASIS)
    K = mat_mul(K, P, BASIS)
    KA = mat_mul(get_inverse(P, BASIS), A_pub, BASIS)
    KA = mat_mul(KA, P, BASIS)

```

```
    solve = to_pari_poly(KA[0][0].polynomial(), 'g')
    ans = int(pari(f'g = ffgen({pari_poly});fflog({solve}, g)'))
    print(ans)
    anss.append(ans)
    moduli.append(2^ii - 1)

B_pub = load_matrix('bob.pub', GF(2))
print(anss)
A_priv = CRT_list(anss, moduli)
shared_secret = B_pub^A_priv

mat_str = ''.join(str(x) for row in shared_secret for x in row)

open('MAT_STR.txt').write(mat_str)

```

USING THE JNF

For now, let's assume we can compute the JNF of the matrices in the challenge and furthermore, that the Jordan form is diagonal (as opposed to block diagonal). Recall that the Jordan normal form of a matrix AA is a (unique) block diagonal matrix JJ such

that J=P−1APJ=P−1AP, where PP is an invertible transformation matrix. This is of particular use to us as we can easily compute/work with powers of matrices once we have the Jordan form:
J2=(P−1AP)(P−1AP)=P−1A2PJ2=(P−1AP)(P−1AP)=P−1A2P
(and in general, Jk=P−1AkPJk=P−1AkP).
In the challenge, we have the generator matrix GG and the public matrices A=GaA=Ga and B=GbB=Gb of Alice and Bob.
Suppose G=P−1JPG=P−1JP where JJ is the Jordan form of GG and PP is the corresponding transformation matrix. Note that J=diag(λ1,…,λN)J=diag(λ1,…,λN) where λiλi are the eigenvalues of GG. Then, A=Ga=P−1JaPA=Ga=P−1JaP, so we have PAP−1=Ja=diag(λa11,…,λanN)PAP−1=Ja=diag(λ1a1,…,λNan). Therefore, we can reduce solving the discrete logarithm problem over matrices to solving the discrete logarithm problem over the underlying group. Specifically, we solve the DLP instance for aiai given λiλi and λaiiλiai. Noting that ai≡a(mod|λi|)ai≡a(mod|λi|), we see that by solving enough instances, we can combine the results with CRT to recover aa completely.

Then I used MAT_STR.txt to get my flag

```python
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Util.number import *
from Crypto.Util.Padding import pad, unpad


mat_str = open('MAT_STR.txt').read()
KEY_LENGTH = 128

key = SHA256.new(data=mat_str.encode()).digest()[:KEY_LENGTH]
flag = {"iv": "43f14157442d75142d0d4993e99a9582", "ciphertext": "22abc3b347ffef55ec82488e5b4a338da5af7ef1918ac46f95029a4d94ace4cb2700fa9aeb31e6a4facee2601e99dabd6f9a81494c55f011e9227c9a6ae8d802"}
cipher = AES.new(key, AES.MODE_CBC, bytes.fromhex(flag['iv']))

pt = cipher.decrypt(bytes.fromhex(flag['ciphertext']))

print(pt)

```

```
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/…/uni/Kryptografia/Diffie-Hellman/Matrix_Revolutions(125p)]
└─$ python3 matrix_revo.py
b'crypto{we_are_looking_for_the_keymaker_478415c4}\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10'
```

FLAG: crypto{we_are_looking_for_the_keymaker_478415c4}

THE END